Reducing synchronization cost in distributed multi-resource allocation problem

Jonathan Lejeune, Luciana Arantes, Julien Sopena, Pierre Sens

Reducing synchronization cost in distributed multi-resource allocation problem

Jonathan Lejeune
Ecole des Mines de Nantes
EMN-Inria, LINA
Nantes, France
Email:jonathan.lejeune@mines-nantes.fr

Luciana Arantes, Julien Sopena, and Pierre Sens
Sorbonne Universités, UPMC Univ Paris 06,
CNRS, Inria, LIP6
F-75005, Paris, France
Email: firstname.lastname@lip6.fr

***Abstract*—Generalized distributed mutual exclusion algorithms allow processes to concurrently access a set of shared resources. However, they must ensure an exclusive access to each resource. In order to avoid deadlocks, many of them are based on the strong assumption of a prior knowledge about conflicts between processes' requests. Some other approaches, which do not require such a knowledge, exploit broadcast mechanisms or a global lock, degrading message complexity and synchronization cost. We propose in this paper a new solution for shared resources allocation which reduces the communication between non-conflicting processes without a prior knowledge of processes conflicts. Performance evaluation results show that our solution improves resource use rate by a factor up to 20 compared to a global lock based algorithm.**

***Keywords*-distributed algorithm, generalized mutual exclusion, multi-resource allocation, drinking philosophers, performance evaluation**

I. Introduction

Processes in distributed and parallel applications require an exclusive access to one or more shared resources. In the case of a single shared resource, one of the standard distributed mutual exclusion algorithms (e.g. [1],[2],[3], [4], [5], [6]) is usually applied in order to ensure that at most one process uses the resource at any time (safety property) and that all requests are eventually satisfied (liveness property). The set of instructions of processes' code that access the shared resource is then called the critical section (CS). However, most of distributed systems such as Clouds or Grids are composed of multiple resources and processes may ask access to several of them simultaneously. Thus, the mutual exclusion principle is extended to several resources but, in this case, a process can access the requested resources only after having obtained the right to access all of them.

Nevertheless, exclusive access to each resource is not enough to ensure the liveness property since deadlock scenarios can take place. In the context of multiple resources allocation, such a problem can happen when two processes are waiting for the release of a resource owned by the other one. This multi-resource problem, also called AND-synchronization, has been introduced by Dijkstra [7] with the dining philosopher problem where processes require the same subset of resources all the time. Later, it was extended by Chandy-Misra [8] to the drinking philosopher problem where processes can require different subset of resources.

In the literature, we distinguish two families of algorithms which solve the multi-resource problem: incremental ([9], [10]) and simultaneous (e.g. [11], [12], [13], [14], [15]). In the first family, a total order is defined for the set of resources and processes must acquire them respecting such an order. In the second one, algorithms propose some mechanisms which allow them to acquire the set of resources atomically without entailing conflicts. On the one hand, many of the proposed solutions of the incremental family consider a priori known *conflict graph* where vertices represent processes and edges model concurrent request to the same resource. However, considering a known and static graph which is a very strong assumption about the application. On the other hand, some solutions of the simultaneous family do not require any knowledge about the conflict graph. Nevertheless, in order to serialize the requests, these solutions have a huge synchronization cost which entails performance degradation of both resource use rate and average waiting time. Other solutions exploit one or several coordinators to order the requests and avoid, thus, deadlocks, but, since they are not fully distributed, they can generate some network contentions when the system load is high. Finally, some algorithms use broadcast mechanisms which render them not scalable in terms of message complexity.

In this paper, we propose a new decentralized approach for locking multiple resources in distributed systems. Our solution does not require the strong hypothesis of a priori knowledge about the conflict graph and does not need any global synchronization mechanism. Moreover, it dynamically re-orders resource requests in order to exploit as much as possible the potential parallelism of non-conflicting requests. Performance evaluation results confirm that our solution improves performance in terms of resources use rate and average request waiting time.

The rest of the paper is organized as follows. Section II discusses some existing distributed algorithms which solve the multi-resource allocation problem. A general outline of our proposal and its implementation are described in sections III and IV respectively. Section V presents performance evaluation results of this implementation by comparing them with two existing solutions of the literature. Finally, Section VI concludes the paper.

II. Related Work

The original mutual exclusion problem was generalized in different ways:

- *group mutual exclusion problem* [16], [17], [18]: a shared resource can be accessed concurrently but only within the same session.
- *k-mutual exclusion problem* [19], [20], [21], [22], [23], [24], [25]: there are several copies (units) of the same critical resource but the access to each one must be exclusive.
- multi-resource allocation problem: single copy of different types of resources. A processes can ask for a set of resources but the access to each copy must be exclusive.

In this paper we will focus on the third problem. Hence, in the rest of this section, we outline the main distributed multi-resource allocation algorithms of the literature. They are basically divided into two families: incremental and simultaneous.

A. Incremental family

In this family, each process locks incrementally a set of required resources, according to a total order defined over the global set of resources. Mutual exclusion to each resource can be ensured with a single-resource mutual exclusion algorithm. However, such a strategy may be ineffective if it presents a domino effect[1] when processes wait for available resources. The domino effect may dramatically reduce the concurrency of non-conflicted nodes and, therefore, hugely degrade resources use rate.

In order to avoid the domino effect, Lynch [9] proposes to color a dual graph of the conflict graph. Then, it is possible to define a partial order over the resources set by defining a total order over the colors set. This partial order reduces the domino effect and improves the parallelism of non-conflicting requests.

Aiming at reducing the waiting time, Styer and Peterson [10] consider an arbitrary coloring (preferably optimized) which also supports request cancellation: a process can release a resource even if it has not use it yet. Such an approach dynamically breaks possible waiting chains.

B. Simultaneous family

In this class of algorithms, resources are not ordered. Algorithms implement some internal mechanisms in order to avoid deadlocks and atomically lock the set of resources required by the process.

Chandy and Misra [8] have defined the drinking philosophers problem where processes (= philosophers) share a set of resources (= bottles). This problem is an extension of the dining philosophers problem where processes share forks. Contrarily to the latter, where a process always asks for the same subset of resources, i.e., the same two forks, the drinking philosopher problem let a process to require a different subset of resources at each new request. The communication graph among processes corresponds to the conflict graph and has to be known in advance. Each process shares a bottle with each of its neighbors. By orienting the conflict graph we obtain a precedence graph. Note that if cycles are avoided in the precedence graph, deadlocks are impossible. It has been shown that the dining philosophers problem respects this acyclicity but it is not the case for the drinking philosophers one. To overcome this problem, Chandy and Misra have applied dining procedures in their algorithms: before acquiring a subset of bottles among its incident edges, a process firstly needs to acquire all the forks shared with its neighbors. Forks can be seen as auxiliary resources that serialize bottle requests in the system and are released when the process has acquired all the requesting bottles. Serialization of requests avoids cycles in the precedence graph and, therefore, deadlocks are avoided. On the other hand, the forks acquisition phase induces synchronization cost.

Ginat et al. [13] have replaced the dining phase of the Chandy-Misra algorithm by logical clocks. When a process asks for its required resources, it timestamps the request with its local logical clock value and sends a message to each neighbor in the conflict graph. Upon receipt of a request, the associate shared bottle is sent immediately if the request timestamp value is smaller than the current clock value of the receiver. The association of a logical clock value and the total order over identifiers of processes defines a total order over requests which prevents deadlocks. However, message complexity becomes high whenever the conflict graph is unknown (equivalent to a complete graph since each process may be in conflict with all the other ones) as the algorithm uses, in this case, a broadcast mechanism.

In [26], Rhee presents a request scheduler where each processes is a manager of a resource. Each manager locally keeps a queue that can be rescheduled according to new pending requests avoiding, therefore, deadlocks. This approach requires several dedicated managers which can become potential bottlenecks. Moreover, the coordination protocol responsible for avoiding deadlocks between managers and application processes is quite costly.

Maddi [14] proposed an algorithm which is based on a broadcast mechanism and each resource is represented by a single token. Each process request is timestamped with the local clock value of the process and broadcast to all other processes. Upon reception, the request is stored in a local queue of the receiver, ordered by the request timestamps. This algorithm can be seen as multiple instances of Susuki-Kasami mutual exclusion algorithm [3], presenting, thus, high messages complexity.

The Bouabdallah-Laforest token-based algorithm [12] is described in more details in this section because it is the closest one to our solution and, therefore, the performance of both algorithms will be evaluated and compared in section V. A single *resource token* and a distributed queue are

[1] A process waits for some resources which are not in use but locked by other processes that wait for acquiring other resources.

assigned to each resource. For having the right to access a resource, a process must acquire the associated *resource token*. Furthermore, before asking for a set of resources, the requester must firstly acquire a *control token*, which is unique in the system. A Naimi-Tréhel based [5] mutual exclusion algorithm is responsible for handling this control token. This algorithm maintains a dynamic distributed logical tree such that the root of the tree is always the last process that will get the token among the current requesting ones. It also keeps a distributed queue of pending requests. The *control token* contains a vector with M entries (the total number of resources of the system) where each entry corresponds to either the *resource token* or the identifier of the latest requester of the resource in question. Thus, when a requesting process receives the *control token*, it acquires all the required resources already included in the *control token* and sends an INQUIRE message to the respective latest requester for each *resource token* which is not in the *control token*. We point out that the *control token* serializes requests, ensuring that a request will be registered atomically in the different distributed waiting queues. Hence, no cycle takes place among all distributed waiting queues. This algorithm presents a good message complexity, but the control token serialization mechanism can induce bottlenecks when the system has few conflicts, i.e., in a scenario where concurrency is potentially high.

III. General outline of our solution

A. Model and assumptions

We consider a distributed system consisting of a finite set Π of reliable N nodes, $\Pi = \{s_1, s_2, ...s_N\}$ and a set of M resources, $\mathcal{R} = \{r_1, r_2, ..., r_M\}$. The set Π is totally ordered by the order relation $\prec$ and $s_i \prec s_j$ iff $i < j$. There is one process per node. Hence, the words node, process, and site are interchangeable. Nodes are assumed to be connected by reliable (neither message duplication nor message loss) and FIFO communication links. Processes do not share memory and communicate by sending and receiving messages. The communication graph is complete, *i.e.*, any node can communicate with any other one. A process can not request a new CS before its previous one has been satisfied. Therefore, there are at most N pending requests. We also assume no knowledge about the conflict graph.

B. Discussion

Similarly to our solution, simultaneous solutions found in the literature do not assume a prior knowledge of the conflict graph. Their control mechanisms totally order requests avoiding, thus, deadlocks. However, they may present poor performance since they induce communication between non conflicting processes which have no need to interact with each other.

On the one hand, Bouabdallah-Laforest [12] is a very effective multi-resource algorithm since it presents logarithmic message complexity. On the other hand, it presents two constraints which degrade resource use rate:

- two non conflicting sites communicate with each other in order to exchange the *control token*, inducing additional cost in terms of synchronization;
- request scheduling is static: it depends only on the acquisition order of the *control token* by the requesting processes. Consequently, a new request is not able to preempt another one which obtained the control token before it, preventing, therefore, a dynamic scheduling which would increase resource use rate.

Hence, our objective is twofold:

- not to use a global lock to serialize requests in order to avoid useless communication between non conflicting processes,
- to schedule requests dynamically.

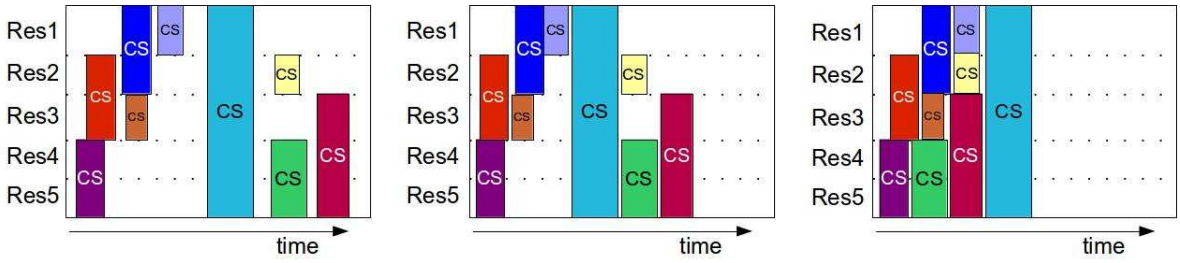
Figure 1 shows, in a system with five shared resources, the impact of our two objectives (lack of global lock and dynamic schedule) on the resource use rate when compared to Bouabdallah-Laforest's algorithm [12] (global lock and static scheduling) and a mofified version of the latter without global lock and static scheduling:

- the lack of global lock reduces the time between two successive conflicting critical sections (Figure 1(b)).
- the dynamic scheduling makes possible the granting of resources to processes in idle time periods (white spaces) where resources are not in use (Figure 1(c)).

C. Suppression of global lock

In order to serialize the requests without using a global lock, we propose a counter mechanism and totally ordering the requests based on the counter values and the identifiers of the nodes.

1) Counter mechanism: The goal of the control token in Bouabdallah-Laforest's algorithm is to provide a unique scheduling order over the whole requesting waiting queues associated to resources. In order to remove this global lock, we have assigned one counter per resource. Each counter provides then a global request order for the resource to which it is related. Hence, there are M counters in the system that should be accessed exclusively, i.e., there is a token associated to each counter whose current value is kept in the respective token. Therefore, a requesting process should firstly obtain, for each requested resource, the current value of the counter of each of these resources. Then, each token holder, related to these counters, atomically increments the respective counter in order to ensure different values at each new request. Once a process has acquired all the required counter values, its request can be associated with a single vector of M integers in the set $\mathbb{N}^M$. Entries of the vector corresponding to non required resources are equal to zero. Consequently, every request is uniquely identified regardless of the time when it has been issued as well as the set of required resources. Then, a process can request its resources independently. Note that this counter mechanism and the exclusive access to a resource are independent: it is always possible to ask for the value of a resource counter while the resource in question is currently in use.



(a) With global lock, static scheduling (b) Without global lock, static scheduling (c) Without global lock, dynamic scheduling

Figure 1. Illustration of the impact of our objectives on the resource use rate

2) Total order of requests: A request req_i issued by the site $s_i \in \Pi$ for a given resource is associated with two pieces of information: the identifier of the requesting process s_i and the respective associated vector $v_i \in \mathbb{N}^M$. Deadlocks are avoided if a total order over requests is defined. To this end, we firstly apply a partial order over the vector values by defining a function $\mathcal{A} : \mathbb{N}^M \rightarrow \mathbb{R}$ which transforms the values of a counter vector in a real value. Since such an approach guarantees just a partial order, we use the identifier of the sites to totally order the requests. Therefore, we define this total order, denoted, $\lhd$ by $req_i \lhd req_j$ iff $\mathcal{A}(v_i) < \mathcal{A}(v_j) \vee (\mathcal{A}(v_i) = \mathcal{A}(v_j) \wedge s_i \prec s_j)$. Thus, if $\mathcal{A}$ returns the same real value for two requests' vector values, the identifiers of the corresponding requesting sites break the tie. Although this mechanism avoids deadlocks by ensuring that all requests can be distinguished, the satisfaction of the algorithm's liveness property depends on the choice of a suitable function $\mathcal{A}$. In other words, $\mathcal{A}$ should avoid starvation by ensuring that every request will have, in a finite time, the smallest real value among all pending requests according to the order $\lhd$. The function $\mathcal{A}$ is a parameter of the algorithm and, basically, defines the scheduling resource policy.

D. Dynamic scheduling

The introduction of a *loan mechanism* into the algorithm could improve the resource use rate. Requested resources are acquired progressively but are actually used once the process got the right to access all of them. Thus, some or even many resources are locked by processes which are not able to use them. Such a behavior reduces the overall resource use rate. The idea of the dynamic scheduling is then to restrict as much as possible the right to access a resource only to critical section execution, i.e., to offer the possibility to lend the right to access a resource to another process. However, for sake of the liveness property, the loan mechanism has to guarantee that eventually a site get back the right, previously acquired, to access the resource. In other words, it must avoids starvation and deadlocks.

1) Starvation avoidance: Since the lending of the right to access a resource will not necessarily ensure that the borrower process will own all the set of resources it needs, starvation problems may occur. To overcome this problem, we propose a simple mechanism by restricting the loan to only one process at a time. We thus guarantee that the lender process will obtain again all the lent resource access rights in a finite time since the critical section time of the borrower is bounded by assumption.

2) Deadlock avoidance: Resources borrowed from multiple processes can lead to cycles in the different resources waiting queues and, therefore, to deadlocks. To avoid it, we propose to restrict the loan to a single site provided that the lender process owns all the resource access rights which are missing to the borrower process. Consequently, upon reception of the rights, the latter can immediately execute its critical section.

IV. Description of the implementation

In this section we describe the principle of our multi-resource allocation algorithm. Due to lack of space, we are not going to present the pseudo-code of it. However, the pseudo-code, the proof of correctness, and a more detailed description of the algorithm are given in [27].

Each resource is associated with a unique token which contains the resource counter. The process that holds the token is the only one which has the right to access and increment the counter value ensuring, therefore, an exclusive access.

Each token is controlled by an instance of a simplified version of the Mueller algorithm [28]. The latter is a prioritized distributed token-based mutual exclusion algorithm that logically organizes the processes in a dynamic tree topology where the root of the tree is the token holder of the corresponding resource. Every token also keeps the queue of pending requests related to the resource it controls.

For instance, in Figure 3, we consider 3 processes (s_1, s_2, and s_3) and 2 resources (r_{red} and r_{blue}). Figure 3(a) shows the initial tree topologies related to each of the resources where s_1 and s_3 hold the token associated with r_{red} and r_{blue} respectively. Notice that s_2 has 2 fathers, s_1 (red tree) and s_3 (blue tree), while s_1 (resp. s_3) has a single father s_3 (resp. s_2) associated with the blue tree (resp. the red tree). In Figure 3(c), the topologies of the trees have changed since s_2 got the two tokens and it is, therefore, the root of both trees.

We should point out that the choice of a prioritized algorithm as Mueller's one makes possible the rescheduling of pending requests of a given resource queue whenever a request, with a higher priority according to the $\triangleleft$ order, regarding this resource, is received.

A. Process states

A process can be in one of the following four states:

- $Idle$: the process is not requesting any resource;
- $waitS$: the process is waiting for the requested counter values;
- $waitCS$: the process is waiting for the right to access all the requested resources.
- $inCS$: the process is using the requested resources (in critical section).

Figure 2 shows the global machine states of a process.

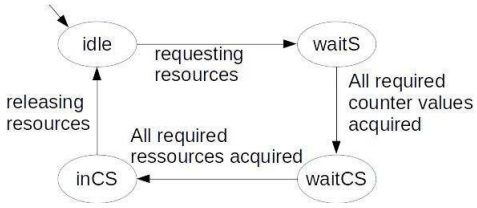


Figure 2. Machine state of a process

B. Messages

We define five types of message:

- $ReqCnt(r, s_{init})$: sent by s_{init} when requesting the current value of the counter associated with r.
- $Counter(r, val)$: sent by the token holder associated with the resource r as a reply to a $ReqCnt$ request. It contains the value val of the r counter that the token holder has assigned to the request in question.
- $ReqRes(r, s_{init}, mark)$: sent whenever s_{init} requests the right to access resource r. The request is tagged with $mark$, the value returned by function $\mathcal{A}$.
- $ReqLoan(r, s_{init}, mark, missingRes)$: sent by s_{init} whenever it requests a loan of resource r tagged with $mark$ value (return of function $\mathcal{A}$). The message also contains the set of missing resources $missingRes$ which s_{init} is waiting for.
- $Token(r, counter, wQueue, wLoan, s_{lender})$: The token message related to resource r which contains the latest value of the associated counter and the waiting queue $wQueue$ of pending requests in increasing order of the respective $marks$. The queue $wLoan$ contains the set of pending requested loans concerning r and, if the latter is currently lent, s_{lender} corresponds to the identifier of the lender site.

Request messages ($ReqCnt$, $ReqRes$, and $ReqLoan$ types) are forwarded from the sender s_{init} till the token holder along the corresponding tree structure while the messages of type $Counter$ and $Token$ are sent directly to the requester.

Note that the graph that represents the tree topology dynamically changes during message forwarding which: (1) may lead to cycles and indefinitely forwarding of requests and (2) starvation problem, i.e., requests that are never satisfied. For avoiding the first problem, we have included in every request message the identifiers of the nodes already visited by the request. For the second one, each site keeps a local history of received request messages, whose obsolete messages can be discarded thanks to a timestamp mechanism.

It is worth also pointing out that in order to reduce the number of messages in our implementation, whenever possible, messages with same type related to the same resource and addressed to the same site can be combined into a single message of this type.

C. The counter mechanism

When process s_i wishes to access a set of resources, it changes its state from $Idle$ to $waitS$. Then, it has to get the current value of the counters associated with all these resources. If s_i already owns the tokens associated with some of the required resources, it reserves to its request the current value of the respective counters and increases them. We should remind that only the token holder (s_i in this case for the tokens it holds) has the right to increase the counters associated with the resources in question. Otherwise, for each missing counter value, s_i sends a $ReqCnt$ message to one of its fathers, i.e., the one which is its father in the corresponding resource tree. It also registers in its local $CntNeeded$ set variable the id. of the missing resources. Process s_i then waits to receive the missing counter values.

When s_j receives the request $ReqCnt$ message for resource r from s_i, if it does not hold the token related to r, it forwards the message to its father which belongs to the r tree. On the other hand, if s_j is the token holder, but does not require r, it directly sends the token to s_i. Otherwise, s_j keeps the token and sends a $Counter$ message, which contains the current value of the counter to s_i and then, increments the counter.

Upon receipt of a $Counter$ message for the resource r, s_i removes r from its $CntNeeded$ set. When $CntNeeded$ becomes empty, s_i has obtained counter values for all the resources it required. Note that these values are uniquely assigned to the requests of s_i. It then changes its state to $waitCS$ and for each of these resources, whose token it does not hold yet, it sends a $ReqRes$ message to the respective father.

Similarly to the $ReqCnt$ message, when receiving a $ReqRes$ for a resource r, process s_j forwards the message to its father if it does not hold the token associated with r. If s_j holds the token and does not require r or is in the $waitS$ state, it sends the token directly to s_i. Otherwise, s_i and s_j are in conflict and it is necessary to take a decision about which of them has the right to the resource r. If s_j is in critical section ($inCS$ state) or if the priority of its request is higher than the s_i's request ($req_j \triangleleft req_i$), it keeps the right. In this case, s_i's request is registered in the r token queue

($wQueue$). Otherwise, s_j has to grant the right to access r to s_i. To this end, it registers its own request in the r token queue ($wQueue$) and sends the token directly to s_i, i.e., a $Token$ message.

When s_i receives a $Token$ message related to r, it makes two updates: (1) it includes r in its set of owned tokens. If r belongs to $CntNeeded$, i.e., s_i has not received all the counter values required in its last request, it registers the current value of the token counter for this request, increments the counter and removes r from $CntNeeded$; (2) Then s_i takes into account pending messages of the local history for the concerned resource: it replies by a $Counter$ message to each site that has issued a $ReqCnt$ message and adds in $wQueue$ (respectively $wLoan$) of the token the information related to $ReqRes$ (resp. $ReqLoan$) messages. Site s_i can enter in critical section ($inCS$ state) if it owns the right to access all the requested resources. If it is not the case, it can change its state to $waitCS$ provided its $CntNeeded$ set is empty (i.e., s_i got all the asked counter values). In this case, s_i sends $ReqRes$ messages for each missing resources. Due to the updates of (2), site s_i has to ensure that its request has the highest priority according to the $\triangleleft$ order. If it is not the case, the token is granted to the site having the highest priority. Site s_i can now potentially satisfy a loan request stored in $wLoan$ concerning the other tokens that it keeps. Finally, s_i can initiate a loan request, if necessary (see section IV-D).

When the process exits the critical section, it dequeues the first element of the waiting queue of all owned resource tokens and sends to their next holder (or potentially the lender site) the associated token. Finally s_i's state becomes $Idle$.

Let's take up the example of Figure 3 with the 3 processes (s_1, s_2 and s_3) and the 2 resources (r_{red} and r_{blue}), where the initial configuration is given in Figure 3(a), that we have previously described. Processes s_1 and s_3 are in critical section accessing r_{red} and r_{blue} respectively. Figure 3(b) shows the messages that processes exchange when s_2 requires both resources. First, s_2 sends to each of its fathers, s_1 (red tree) and s_3 (blue tree), a $ReqCnt$ request in order to obtain the associated current counter values. When s_2 has received the two requested counter values, it sends $ReqRes$ messages along the trees asking for respective resources. Upon exiting the critical sections s_1 and s_3 respectively send r_{red} token and r_{blue} token to s_2, which can thus enter the critical section once it received both tokens. The final configuration of the logical trees is shown in Figure 3(c).

D. The loan mechanism

The execution of a loan depends on some conditions related to both the lender and the borrower sites:

- Upon receipt of a token, process s_i can request a loan provided it is in the $waitCS$ state (i.e., it got all the needed counter values) and the number of missing resources is smaller or equal to a given threshold. If it is the case, s_i sends a $ReqLoan$ message to the respective father of the missing resources trees. Similarly to a $ReqRes$ message, a $ReqLoan$ message for a resource is forwarded till the token holder associated with this resource.
- When receiving a $ReqLoan$ message for resource r, the token holder s_j first checks if the loan is possible. All required tokens in the message ($missingRes$ set) can be lent if the following conditions are met:
  - s_j owns all the requested resources (indicated in the $ReqLoan$ message by the $missingRes$ set);
  - none of the resources owned by s_j is a loan;
  - s_j has not lent resources to another site;
  - s_j is not in critical section
  - s_i's request has a higher priority than s_j's request if both have sent a loan request for their current CS request.

If the loan is feasible, the tokens associated with the resources are sent to s_i with s_{lender} equals to s_j. On the other hand, if s_j does not require the resource of the request or is in $waitS$ state, it sends the token directly to the borrower site s_i. Otherwise, i.e., one or more of the above conditions were not satisfied, the loan request is included in the $wLoan$ of the corresponding token to be potentially satisfied later upon receipt of new tokens.

When s_i receives borrowed tokens and if it does not enter in critical section (e.g., if it has yield other tokens for higher priority requests in the meantime), then the loan request fails. Consequently, the loan request is canceled and s_i immediately returns borrowed tokens to s_{lender}, avoiding, therefore, an inconsistent state where a site owns borrowed and unused tokens.

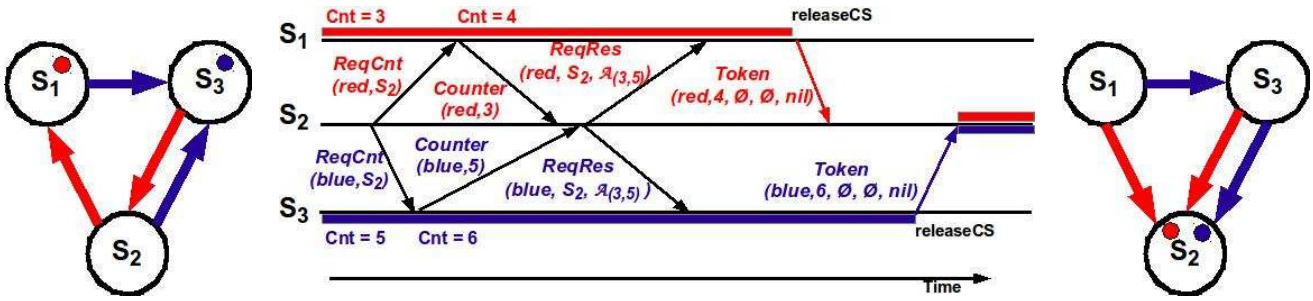
Finally, when exiting the critical section, s_i sends back these tokens directly to s_j.

E. Optimizations

1) Synchronisation cost reduction of single resource requests: It is possible to reduce the synchronization cost of requests requiring a single resource by directly changing the state of the requester from $Idle$ to $waitCS$. Since such requests require only one counter, stored in the token, the root site of the corresponding tree is able to apply $\mathcal{A}$ and then consider the $ReqCnt$ message as a $ReqRes$ message. Hence, such an optimization reduces messages exchanges.

2) Reduction of ReqRes messages: Once a process s_i gets all the requested resource counter values, it sends, for each of these resources, a $ReqRes$ message that will travel along the corresponding tree till the token holder (root site). The number of these forward messages can be reduced by:

- shortcuting the path between the requesting site s_i and the root site s_j: upon receipt of a $Counter$ message from s_j, s_i sets its father pointer to s_j since the latter is the last token owner from the viewpoint of s_i.
- stopping forwarding before the message reaches the root site. When receiving a $ReqRes$ message for a resource r, a process s_j does not forward the message if (1) it is in the $waitCS$ state, also requires r, and its



(a) Initial state (b) Execution (c) Final state

Figure 3. Execution example

request has a higher precedence than s_i's request or (2) s_j has lent the token. If one of these two conditions is met, s_j knows that it will get the token corresponding to r before s_i. The request of s_i will eventually be stored in the waiting queue $wQueue$ of the token.

V. Performance Evaluation

In this section, we present some evaluation results comparing our solution with two other algorithms:

- An algorithm, which we have denoted **incremental algorithm** which uses M instances of the Naimi-Tréhel algorithm [6], one of the most efficient mutual exclusion algorithm thanks to its messages complexity comprised between $\mathcal{O}(LogN)$ and $O(1)$
- The **Bouabdallah-Laforest** algorithm [12] (see Section II).

In order to show the impact of the loan mechanism, we consider two versions of our algorithm named **Without loan** and **With loan** which respectively disable and enable the loan mechanism. In the latter, a site asks for a loan when it has just one missing requesting resource.

We are interested in evaluating the following two metrics: (1) resource use rate and (2) the waiting time to have the right to use all the requested resources, i.e., the right to execute the critical section.

As previously explained, our algorithm requires a function $\mathcal{A}$ as input. For performance evaluation, our chosen function $\mathcal{A}$ computes the average of non null values of the counter vector. This function avoids starvation because counter values increase at each new issued request which implies that the minimum value returned by $\mathcal{A}$ increases at each new request. Thus, the liveness property is ensured. We should emphasize that the advantage of this approach lies in the fact that starvation is avoided only by calling the function and does not induce any additional communication cost.

A. Experimental testbed and configuration

The experiments were conducted on a 32-nodes cluster with one process per node. Therefore, the side effect due to the network is limited since there is just one process per network card. Each node has two 2.4GHz Xeon processors and 32GB of RAM, running Linux 2.6. Nodes are linked by a 10 Gbit/s Ethernet switch. The algorithms were implemented using C++ and OpenMPI. An experiment is characterized by:

- N: number of processes (32 in our experiments).
- M: number of total resources in the system (80 in our experiments).
- α: time to execute the critical section (CS) (4 possible values : 5 ms, 15 ms, 25 ms and 35 ms according to the number of asked resources).
- β: mean time interval between the release of the CS by a node and the next new request issued by this same node.
- γ: network latency to send a message between two nodes (around 0,6 ms for our experiments).
- ρ: the ratio $\beta/(\alpha+\gamma)$, which expresses the frequency with which the critical section is requested. The value of this parameter is inversely proportional to the load: a low value implies a high request load and vice-versa.
- ϕ: the maximum number of resources that a site can ask in a single request which ranges for 1 and M. The greater the value of this parameter, the lower the potential parallelism of the application and thus, the higher the probability to have conflicting requests.

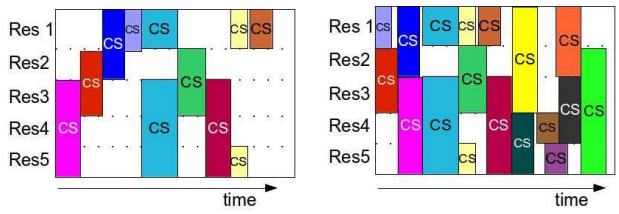
At each new request, a process chooses x resources. The critical section time of the request depends on the value of x: the greater the value, the higher the probability of a long critical section time since a request requiring a lot of resources is more likely to have a longer critical section execution time.

For each metric, we show performance results corresponding to both medium and high load scenarios.

B. Resource use rate

This metric expresses the percentage of time that resources are in use (e.g., 100 % means that all resources are in use during the whole experiment). It can be seen as the percentage of colored area in the diagrams of Figure 4. We can observe that resources are used more effectively in the example of execution of Figure 4(b) than in the example of Figure 4(a), i.e., the former presents fewer white areas.

By varying ϕ, we show in Figure 5 the impact of the number of asked resources within a request, denoted *request*



(a) Inefficient execution (b) Efficient execution

Figure 4. Illustration of the metric of resource use rate

size, on the resource use rate in the case of medium (figure 5(a)) and high (figure 5(b)) loads. The request size x may be different for each new request and it is chosen according to a uniform random law from 1 to ϕ.

In addition to the considered algorithms, we have included in both figures a fifth curve which represents a distributed scheduling algorithm executed on a single shared-memory machine with a global waiting queue and no network communication. The aim of such a curve is the evaluation of the synchronization cost of the different algorithms since it is a resource scheduling algorithm without any synchronization.

Overall, in both figures, whenever ϕ increases, the resource use rate increases too. When the request size is minimal, the maximal number of resources in use is equal to the number of processes N which is smaller than the number of the total resources M. On the other hand, when the average request size increases, each critical section execution concerns a larger number of resources and, therefore, the resource use rate increases.

Note that in high load scenario (Figure 5(b)) the shape of the curve of the scheduling algorithm without synchronization firstly increases, then decreases till a threshold value at $\phi = 20$, and finally increases again. The curve has such a shape due to a threshold effect. In the first rise of the curve, the probability of having conflicts is small compared to both the request size and the difference between N and M. After $\phi = 4$, the number of conflicts starts to increase and the drop that follows is a consequence of the serialization of conflicting requests. Finally, when ϕ is greater than 20, the probability of having requests conflicts is maximum but each critical section access requires a lot of resources which increases the global use rate. Hence, the subsequent rise of the curve is not caused by the increase of non-conflicting requests concurrency, but by the increase of requests' size.

We should also point out that, when the average request size increases, the shapes of the resource use rate curves of the different algorithms are not the same. For the **incremental algorithm**, the resource use rate decreases and stabilizes since this algorithm does not benefit from the increase in the request size due to the domino effect (see II-B). The resource use rate of the **Bouabdallah-Laforest** algorithm increases regularly. Although this algorithm is very disadvantaged by the global lock bottleneck whenever there are few conflicts (especially in high load), its use rate increases with the average request size. We observe that in this algorithm the resource use rate increases faster because it can take advantage of concurrent requests. However, it is not as much effective as our algorithms: independently of the request size, the latter present a higher resource use rate than the former, whose performance is affected by the bottleneck of its control token as well as its static scheduling approach. Notice that, depending on the request size, our algorithms have resource use rate values from 0.4 to 20 times higher than Bouabdallah-Laforest algorithm.

The curves related to the resource use rate of **our two algorithms** have the same shape than the one of the scheduling without synchronization. When the loan mechanism is enabled, the respective algorithm presents a higher resource use rate in high load scenario when the request size lies between 4 and 16 (improvement of up to 15%). Such a behavior shows that the loan mechanism is effective in reducing the negative effect of conflicts induced by medium requests and does not degrade performance when request size is big.

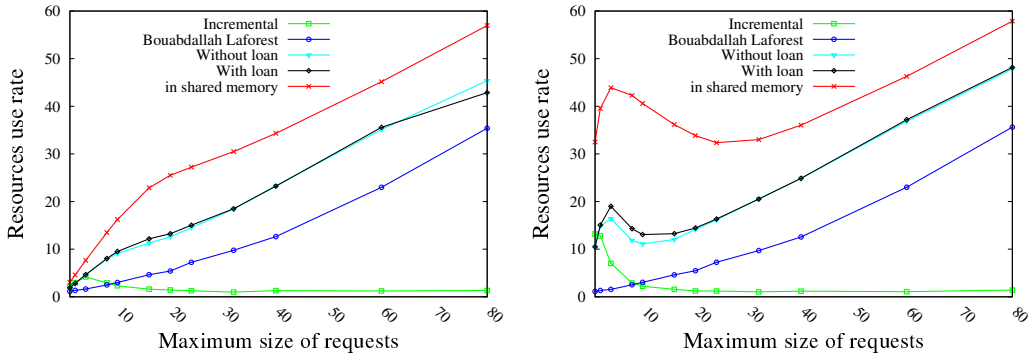
C. Average waiting time

In this section we study the average waiting time of a request which corresponds to the interval from the time the request was issued till the time when the requesting process got the right to access the resources whose identifiers are in the request.

For both high and medium loads, Figures 6 and 7 show the average waiting time for processes to enter in critical section, respectively considering a small ($\phi = 4$) and the highest ($\phi = 80$) maximum request size. Figure 7 presents in more details the waiting time of different request sizes. We have not included in the figures the performance of the incremental algorithm because it is strongly disadvantaged by the domino effect: the average waiting time was too high compared to the experiment duration.

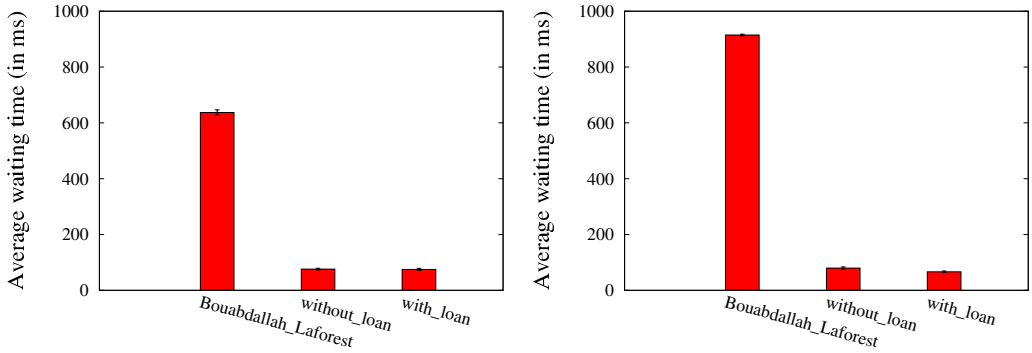
We can note in Figures 6(a) and 6(b) that our algorithms have a lower average waiting time than the Bouabdallah-Laforest algorithm when request size is small (around 11 times lower in high load and 8 times lower in medium load). Such a behavior confirms that our algorithms benefit from its lower synchronization cost. We also observe an improvement of 20% when the loan mechanism is activated in the high load scenario which is consistent with the previous figures related to resource use rate.

On the other hand, contrarily to our algorithms, both the waiting time and the standard deviation of Bouabdallah-Laforest algorithm do not vary much when request size varies, as shown in Figures 7(a) and 7(b). Although our algorithm is the most efficient, its scheduling penalizes requests of small size. We can observe in the same figures that the average waiting time of small requests is the highest one as well as the respective standard deviation. Indeed, due to our chosen scheduling policy, i.e., our function $\mathcal{A}$, the access order of a single resource request depends on the value of the corresponding counter. In other words, the vector value average returned by the function concerns,



(a) Medium load (b) High load

Figure 5. Impact of request size over resource use rate



(a) Medium load (b) High load

Figure 6. Average waiting time ($\phi = 4$)

in this case, just one counter value which is increased according to the frequency with each the associated resource is required: a highly requested resource will have a higher counter value when compared to other ones which are less requested.

VI. Conclusion and Future Work

We have presented in this paper a new distributed algorithm to exclusively allocate a set of different resources. It does not require a prior knowledge about the graph of conflicts and reduces communication between non conflicting processes since it replaces a global lock by a counter mechanism. The totally order of requests can be ensured with the definition of a function $\mathcal{A}$, given as input parameter of the algorithm. Performance evaluation results confirm that the counter mechanism improves the resource use rate and reduces the average waiting time. However, it can not completely avoid the domino effect which increases the waiting time of pending requests. To overcome this drawback, we have include in the algorithm a loan mechanism that dynamically reschedules pending requests, reducing the probability that the domino effect takes place.

Since our solution limits communication between non conflicting processes, it would be interesting to evaluate our algorithm on a hierarchical physical topology such as Clouds, grids and large clusters. The lack of global lock of our algorithm would avoid useless communication between two distant geographic sites reducing, therefore, requests waiting time when compared to other control token based multi-resource algorithms. Performance results show that initiating a loan request when a process misses just one resource (threshold =1) improves use rate in scenarios with medium size requests. Thus, it would be interesting to evaluate the impact of this threshold on other metrics.

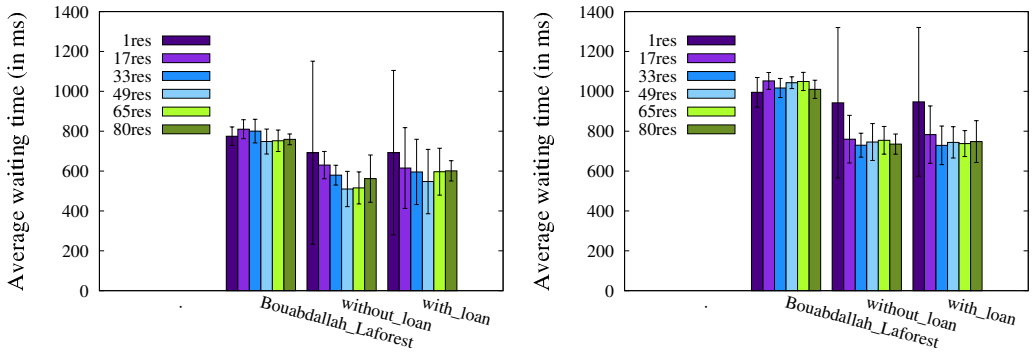
VII. Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the Inria ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[2] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Commun. ACM*, vol. 24, pp. 9–17, January 1981.



(a) Medium load

(b) High load

Figure 7. Average waiting time to get a given number of resources ($\phi = 80$)

[3] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," *ACM Trans. Comput. Syst.*, vol. 3, no. 4, pp. 344–349, 1985.

[4] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," *ACM Trans. Comput. Syst.*, vol. 7, no. 1, pp. 61–77, 1989.

[5] M. Naimi and M. Trehel, "How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion," in *WDAG*, 1987, pp. 155–166.

[6] ——, "An improvement of the log(n) distributed algorithm for mutual exclusion," in *ICDCS*, 1987, pp. 371–377.

[7] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, pp. 115–138, 1971.

[8] K. M. Chandy and J. Misra, "The drinking philosopher's problem," *ACM Trans. Program. Lang. Syst.*, vol. 6, no. 4, pp. 632–646, 1984.

[9] N. A. Lynch, "Upper bounds for static resource allocation in a distributed system," *J. Comput. Syst. Sci.*, vol. 23, no. 2, pp. 254–278, 1981.

[10] E. Styer and G. L. Peterson, "Improved algorithms for distributed resource allocation," in *PODC*, 1988, pp. 105–116.

[11] B. Awerbuch and M. Saks, "A dining philosophers algorithm with polynomial response time," in *FoCS, 1990. Proceedings., 31st Annual Symposium on*, oct 1990, pp. 65 –74 vol.1.

[12] A. Bouabdallah and C. Laforest, "A distributed token/based algorithm for the dynamic resource allocation problem," *Operating Systems Review*, vol. 34, no. 3, pp. 60–68, 2000.

[13] D. Ginat, A. U. Shankar, and A. K. Agrawala, "An efficient solution to the drinking philosophers problem and its extension," in *WDAG (Disc)*, 1989, pp. 83–93.

[14] A. Maddi, "Token based solutions to m resources allocation problem," in *SAC*, 1997, pp. 340–344.

[15] V. C. Barbosa, M. R. F. Benevides, and A. L. O. Filho, "A priority dynamics for generalized drinking philosophers," *Inf. Process. Lett.*, vol. 79, no. 4, pp. 189–195, 2001.

[16] Y.-J. Joung, "Asynchronous group mutual exclusion (extended abstract)," in *PODC*, 1998, pp. 51–60.

[17] V. Bhatt and C. Huang, "Group mutual exclusion in $O(\log n)$ RMR," in *PODC 2010, Zurich, Switzerland, July 25-28, 2010*, 2010, pp. 45–54.

[18] Aoxueluo, W. Wu, J. Cao, and M. Raynal, "A generalized mutual exclusion problem and its algorithm," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, 2013, pp. 300–309.

[19] K. Raymond, "A distributed algorithm for multiple entries to a critical section," *Inf. Process. Lett.*, vol. 30, no. 4, pp. 189–193, 1989.

[20] M. Raynal, "A distributed solution to the k-out of-m resources allocation problem," in *ICCI*, 1991, pp. 599–609.

[21] M. Naimi, "Distributed algorithm for k-entries to critical section based on the directed graphs," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 4, pp. 67–75, Oct. 1993.

[22] N. S. DeMent and P. K. Srimani, "A new algorithm for k mutual exclusions in distributed systems," *Journal of Systems and Software*, vol. 26, no. 2, pp. 179–191, 1994.

[23] R. Satyanarayanan and C. R. Muthukrishnan, "Multiple instance resource allocation in distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 23, no. 1, pp. 94–100, 1994.

[24] S. Bulgannawar and N. H. Vaidya, "A distributed k-mutual exclusion algorithm," in *ICDCS*, 1995, pp. 153–160.

[25] V. A. Reddy, P. Mittal, and I. Gupta, "Fair k mutual exclusion algorithm for peer to peer systems," in *ICDCS*, 2008, pp. 655–662.

[26] I. Rhee, "A modular algorithm for resource allocation," *Distributed Computing*, vol. 11, no. 3, pp. 157–168, 1998.

[27] J. Lejeune, L. Arantes, J. Sopena, and P. Sens, "Reducing synchronization cost in distributed multi-resource allocation problem," Inria, Research Report RR-8689, Feb. 2015. [Online]. Available: https://hal.inria.fr/hal-01120808

[28] F. Mueller, "Priority inheritance and ceilings for distributed mutual exclusion," in *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, 1999, pp. 340 –349.